DOI:
10.1016/j.tcs.2016.11.035

Document Version
Peer reviewed version

Efficient Algorithms for Shortest Partial Seeds in Words

Tomasz Kociumaka[a], Solon P. Pissis[b], Jakub Radoszewski[a,b], Wojciech Rytter[a], Tomasz Waleń[a]

[a] *Institute of Informatics, University of Warsaw, Warsaw, Poland*
[b] *Department of Informatics, King's College London, London, UK*

Abstract

A factor u of a word w is a *cover* of w if every position in w lies within some occurrence of u in w. A factor u is a *seed* of w if it is a cover of a superstring of w. Covers and seeds extend the classical notions of periodicity. We introduce a new notion of *α-partial seed*, that is, a factor covering as a seed at least α positions in a given word. We use the Cover Suffix Tree, recently introduced in the context of *α-partial covers* (Kociumaka et al., *Algorithmica*, 2015); an $\mathcal{O}(n \log n)$-time algorithm constructing such a tree is known. However, it appears that partial seeds are more complicated than partial covers—our algorithms require algebraic manipulations of special functions related to edges of the modified Cover Suffix Tree and the border array. We present a procedure for computing shortest α-partial seeds that works in $\mathcal{O}(n)$ time if the Cover Suffix Tree is already given.

This is a full version, which includes all the proofs, of a paper that appeared at CPM 2014 [1].

Keywords: algorithms on strings, quasiperiodicity, suffix tree, cover, seed

1. Introduction

Periodicity in words is a fundamental topic in combinatorics on words and string algorithms (see, e.g., [2]). The concept of quasiperiodicity generalizes the notion of periodicity [3], and it allows detecting repetitive structure of words which cannot be found using the classic characterizations of periods. Several types of quasiperiods have been introduced; each of them reveals slightly different kind of repetitive structures.

The best-known type of quasiperiodicity is the *cover* of a word. A factor u of a word w is said to be a cover of w if every position in w lies within some occurrence of u in w; we also say that w is covered by u. An extension of the notion of a cover is that of a *seed*. In this case, the positions considered to be covered by a factor u also include those within overhanging occurrences of u. Equivalently, a factor u is a seed of w if and only if w is a factor of a word y covered by u.

Several efficient algorithms for computation of covers and seeds have been developed since the introduction of these notions in early 1990s. The first one, by Apostolico et al. [4], is a linear-time procedure finding the shortest cover of a word. Moore and Smyth [5, 6] showed that all covers can be determined in the same time complexity. Linear-time algorithms providing yet more complete characterizations of covers by so-called cover arrays were given in [7, 8]. Seeds were introduced by Iliopoulos, Moore, and Park [9], who presented an $\mathcal{O}(n \log n)$-time algorithm identifying those factors. This result was much later improved by Kociumaka et al. [10], who gave a complex linear-time algorithm.

Even though quasiperiodicities give much more flexibility than the classic periodic structures, it remains unlikely that an arbitrary word has a cover or a seed other than the whole word itself. Due to this reason, relaxed variants of quasiperiodicity have been introduced. One of the ideas, resulting in the notions of *approximate covers* [11] and *approximate seeds* [12], is to require that each position lies within an approximate occurrence of the corresponding quasiperiod. Another approach, introduced recently in [13], yields the notion

Email addresses: kociumaka@mimuw.edu.pl (Tomasz Kociumaka), solon.pissis@kcl.ac.uk (Solon P. Pissis), jrad@mimuw.edu.pl (Jakub Radoszewski), rytter@mimuw.edu.pl (Wojciech Rytter), walen@mimuw.edu.pl (Tomasz Waleń)

of *partial covers*—factors required to cover a certain number of positions of a word. Partial covers generalize the earlier notion of *enhanced covers* [14], which are additionally required to be borders (both prefixes and suffixes) of the word. In this paper, we extend the ideas behind partial covers and introduce the concept of a *partial seed*.

The *cover index* $\mathcal{C}(u, w)$ of a factor u in a word w has been defined in [13] as the number of positions in w covered by (full) occurrences of u in w. The word u is called an *α-partial cover* of w if $\mathcal{C}(u, w) \geq \alpha$.

We call a non-empty prefix of w that is also a proper suffix of u a *left-overhanging* occurrence of u in w. Symmetrically, a non-empty suffix of w which is a proper prefix of u is called a *right-overhanging* occurrence. The *seed index* of u in w, denoted as $\mathcal{S}(u, w)$, is the number of positions in w covered by full, left-overhanging, or right-overhanging occurrences of u in w. We say that u is an *α-partial seed* of w if $\mathcal{S}(u, w) \geq \alpha$. If the word w is clear from the context, we use the simpler notation $\mathcal{C}(u)$ and $\mathcal{S}(u)$ instead of $\mathcal{C}(u, w)$ and $\mathcal{S}(u, w)$, respectively.

Example 1. *For $w =$ `aaaabaabaaaaaba`, see also Fig. 1, the seed indices of sample factors are as follows:*

$$\mathcal{S}(\texttt{abaa}) = 12,\ \mathcal{S}(\texttt{aba}) = 10,\ \mathcal{S}(\texttt{ab}) = 7,\ \mathcal{S}(\texttt{a}) = 12.$$

```
a b a a         a b a a       a b a a
  a b a a   a b a a         a b a a
    a a a a b a a b a a a a a b a
```

Figure 1: The positions covered by `abaa` as a partial seed of w are underlined. The word `abaa` is a 12-partial seed of w; it has four overhanging occurrences and two full occurrences. Note that `a` is the shortest 12-partial seed of w.

We study the following two related problems:

PartialSeeds
Input: a word w of length n and a positive integer $\alpha \leq n$
Output: all shortest factors u such that $\mathcal{S}(u, w) \geq \alpha$

LimitedLengthPartialSeeds
Input: a word w of length n and an interval $[\ell, r]$
Output: a factor u, $|u| \in [\ell, r]$, which maximizes $\mathcal{S}(u, w)$

In [13] a data structure called the *Cover Suffix Tree* and denoted by $CST(w)$ was introduced. For a word w of length n the size of $CST(w)$ is $\mathcal{O}(n)$ and the construction time is $\mathcal{O}(n \log n)$.

Our results. In this article, we extend the Cover Suffix Tree to support queries concerning partial seeds.

Theorem 2. *Given $CST(w)$, the* LimitedLengthPartialSeeds *problem can be solved in $\mathcal{O}(n)$ time.*

By applying binary search, Theorem 2 implies an $\mathcal{O}(n \log n)$-time solution to the PartialSeeds problem. However, the running time can be improved to $\mathcal{O}(n)$, provided that $CST(w)$ is given in advance.

Theorem 3. *Given $CST(w)$, the* PartialSeeds *problem can be solved in $\mathcal{O}(n)$ time.*

Our solution for the PartialSeeds problem can also recover *all* the shortest factors u of w that satisfy the condition $\mathcal{S}(u, w) \geq \alpha$.

Structure of the paper. In Section 2, we introduce basic notation related to words and suffix trees, and we recall the Cover Suffix Tree (CST). Next, in Section 3, we extend CST to obtain its counterpart suitable for computation of partial seeds, which we call the *Seed Suffix Tree* (SST). In Section 4, we introduce two abstract problems formulated in terms of simple functions which encapsulate the intrinsic difficulty of the PartialSeeds and LimitedLengthPartialSeeds problems. Solutions to these problems, presented in the following Section 5, essentially constitute the most involved part of our contribution. We summarize our results in the Conclusions section.

2. Preliminaries

Let us fix a word w of length $|w| = n$ over a totally ordered alphabet Σ. We assume that the letters of w are numbered from 1 to n. By $w[i\,..\,j]$ we denote a *factor* of w being a subsequence of letters of w starting at the position i and ending at the position j. For a factor v of w, by $Occ(v)$ we denote the set of positions where occurrences of v in w start. By $first(v)$ and $last(v)$ we denote $\min Occ(v)$ and $\max Occ(v)$, i.e., the positions of the leftmost and the rightmost occurrence of v in w, respectively.

Factors $w[1\,..\,i]$ are called *prefixes* of w, and factors $w[i\,..\,n]$ are called *suffixes* of w. Words shorter than w that are both prefixes and suffixes of w are called *borders* of w. By $\beta(w)$ we denote the length of the longest border of w. The border array $\beta[1\,..\,n]$ and reverse border array $\beta^R[1\,..\,n]$ of w are defined as follows: $\beta[i] = \beta(w[1\,..\,i])$ and $\beta^R[i] = \beta(w[i\,..\,n])$. The arrays β, β^R can be constructed in $\mathcal{O}(n)$ time [15].

The *suffix tree* of w, denoted by $ST(w)$, is the compacted suffix trie of w in which only branching and terminal nodes are stored explicitly, while the remaining nodes are implicit; see [15, 16]. We identify the nodes of $ST(w)$ with the factors of w that they represent. An *augmented* suffix tree may explicitly represent some additional nodes, called *extra* nodes. For an explicit node $v \neq \varepsilon$, we set $path(v) = (v_0, v_1, \ldots, v_k)$ where $v_0 = v$ and $v_1, \ldots, v_k$ are the implicit nodes on the path going upwards from v to its nearest explicit ancestor. For example, in the right tree in Fig. 3, we have $path(v) = (v, v_1, v_2, v_3, v_4, v_5)$. We define the *locus* of a factor v' of w as a pair (v, j) such that $v' = v_j$ where $v_j \in path(v)$. The locus is used to refer the node representing v', irrespective of whether it is explicit or implicit.

The Cover Suffix Tree, introduced in [13], is an augmented version of a suffix tree. It allows to efficiently compute $\mathcal{C}(v)$ for any explicit or implicit node, as shown in the following theorem:

Theorem 4 ([13])**.** *Let w be a word of length n. There exists an augmented suffix tree of size $\mathcal{O}(n)$, such that for each edge path(v) we have $\mathcal{C}(v_j) = c(v) - j\Delta(v)$ for some positive integers $c(v)$ and $\Delta(v)$. Such a tree together with the values $c(v)$ and $\Delta(v)$, denoted as $CST(w)$, can be constructed in $\mathcal{O}(n \log n)$ time and $\mathcal{O}(n)$ space.*

Actually, [13] provides explicit formulae for $c(v)$ and $\Delta(v)$ in terms of $Occ(v)$. Their form is not important here; the only property that we use in this contribution is the following:

Property 5. *For every node v of $CST(w)$, $1 \leq \Delta(v) \leq |Occ(v)|$.*

3. Seed Index and Seed Suffix Tree

The cover index $\mathcal{C}(v)$ of any factor v of w can be easily derived from the Cover Suffix Tree $CST(w)$. In order to determine the seed index $\mathcal{S}(v)$, we also need to take into account positions covered by left-overhanging and right-overhanging occurrences of v. The following fact provides a suitable formula based on the border array β and the reverse border array β^R; see also Fig. 2.

Fact 6. *For any factor v of w, the seed index can be expressed as $\mathcal{S}(v) = \mathcal{C}(v) + Left\mathcal{S}(v) + Right\mathcal{S}(v)$, with*

$$Left\mathcal{S}(v) = \min(\beta[first(v) + |v| - 1], first(v) - 1),$$
$$Right\mathcal{S}(v) = \min(\beta^R[last(v)], n - |v| + 1 - last(v)).$$

Proof. The cover index $\mathcal{C}(v)$ counts all positions covered by full occurrences of v. We claim that the remaining positions covered by left-overhanging occurrences are counted by $\mathit{Left}\mathcal{S}(v)$. Let $p = \mathit{first}(v) + |v| - 1$. Note that $w[1\,..\,p]$ has v as a suffix and it does not contain any other full occurrence of v. Consequently, $\beta[p]$ is the length of the longest left-overhanging occurrence of v. This occurrence covers $\beta[p]$ positions, but, among them, positions greater than or equal to $\mathit{first}(v)$ are already covered by a full occurrence of v. The term $\mathit{Right}\mathcal{S}(v)$ has a symmetric interpretation. □

The Cover Suffix Tree $CST(w)$ introduces some extra nodes to the suffix tree $ST(w)$ thanks to which the cover index $\mathcal{C}(v_j)$ on each edge becomes a linear function: $\mathcal{C}(v_j) = c(v) - j\Delta(v)$. The formula for the seed index $\mathcal{S}(v_j)$ is more complex. However, if we make some further nodes explicit, then $\mathcal{S}(v_j)$ becomes a relatively simple function. We call the resulting tree the *Seed Suffix Tree*, denoted as $SST(w)$; see Fig. 3.

Lemma 7. *Let w be a word of length n. There is an augmented suffix tree $SST(w)$ in which each explicit node v has an associated function $\phi_v(x) = a_v x + b_v + \min(c_v, \beta[x])$ and an integer r_v such that for all nodes $v_j \in path(v)$ we have $\mathcal{S}(v_j) = \phi_v(r_v - j)$. Additionally, $0 \le a_v \le |Occ(v)|$. Moreover, the tree $SST(w)$ has $\mathcal{O}(n)$ nodes and, together with the border array β and tuples (a_v, b_v, c_v, r_v) representing ϕ_v, it can be constructed in $\mathcal{O}(n)$ time given $CST(w)$.*

Proof. Consider an edge $path(v) = (v_0, \ldots, v_k)$. For every node v_j, $0 \le j \le k$, we obviously have $\mathit{first}(v_j) = \mathit{first}(v)$, $\mathit{last}(v_j) = \mathit{last}(v)$, and $|v_j| = |v| - j$. Hence, by Fact 6:

$$\begin{aligned}
\mathcal{C}(v_j) &= c(v) - j\Delta(v)\\
\mathit{Left}\mathcal{S}(v_j) &= \min(\beta[\mathit{first}(v) + |v| - j - 1], \mathit{first}(v) - 1)\\
\mathit{Right}\mathcal{S}(v_j) &= \min(\beta^R[\mathit{last}(v)], n - |v| + j + 1 - \mathit{last}(v)).
\end{aligned}$$

We consider the function $\mathcal{S}(v_j) = \mathcal{C}(v_j) + \mathit{Left}\mathcal{S}(v_j) + \mathit{Right}\mathcal{S}(v_j)$. Recall that $\mathcal{C}(v_j)$ is a linear function of j. Also, $\mathit{Right}\mathcal{S}(v_j)$ is relatively simple: it either already is a linear function in the whole range $[0\,..\,k]$, or it becomes one in both parts of $[0\,..\,k]$ once we split this interval at an index $j \in [0\,..\,k]$ such that $\beta^R[\mathit{last}(v)] = n - |v| + j + 1 - \mathit{last}(v)$. We subdivide each edge $path(v)$ at v_j if such an index j exists. Note that we can easily determine the values c and Δ for newly created extra nodes. In total, we introduce at most $\mathcal{O}(n)$ further extra nodes (at most one per edge of $CST(w)$), so the resulting tree $SST(w)$ has $\mathcal{O}(n)$ explicit nodes.

Define

$$r_v = \mathit{first}(v) + |v| - 1.$$

It remains to show that after these transformations it holds that $\mathcal{S}(v_j) = \phi_v(r_v - j)$ and the coefficients of ϕ_v can be efficiently computed. Setting $x = r_v - j = \mathit{first}(v) + |v| - j - 1$, we have

$$\begin{aligned}
\mathcal{C}(v_j) &= x\Delta(v) + c(v) - (\mathit{first}(v) + |v| - 1)\Delta(v) \quad \text{and}\\
\mathit{Left}\mathcal{S}(v_j) &= \min(\beta[x],\ \mathit{first}(v) - 1) \quad \text{and}\\
\mathit{Right}\mathcal{S}(v_j) &= \beta^R[\mathit{last}(v)] \quad \textbf{or} \quad \mathit{Right}\mathcal{S}(v_j) = -x + \mathit{first}(v) - \mathit{last}(v) + n.
\end{aligned}$$

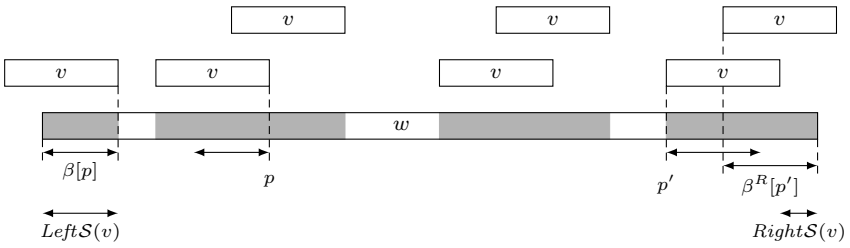

Figure 2: The positions contained in $\mathcal{S}(v)$ are marked grey. In this case, $\mathit{Left}\mathcal{S}(v) = \beta[\mathit{first}(v) + |v| - 1]$ and $\mathit{Right}\mathcal{S}(v) = n - |v| + 1 - \mathit{last}(v)$.

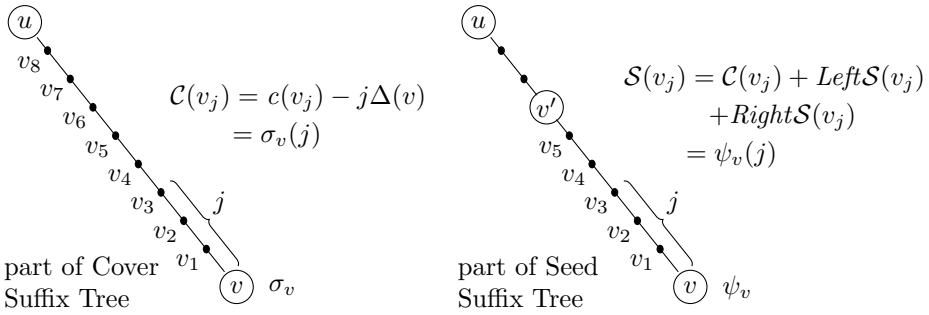

Figure 3: In $CST(w)$, each explicit node stores a constant-space description of a linear function σ_v, which gives the values of $\mathcal{C}(v_j)$ for implicit nodes on the edge from v upwards. In $SST(w)$, there is a corresponding function ψ_v which is a combination of the linear function σ_v and two functions depending on border arrays. With a suitable linear transformation of the argument j, the function $\psi_v(j)$ gets replaced by a function $\phi_v(x)$, which admits a more convenient representation. When transforming $CST(w)$ to $SST(w)$, some implicit nodes (v' on the figure) are made explicit to guarantee that ϕ_v has a simple form.

Thus:

$$a_v = \Delta(v) \quad \text{and} \quad c_v = \mathit{first}(v) - 1 \quad \text{and}$$
$$b_v = \beta^R[\mathit{last}(v)] + c(v) - (\mathit{first}(v) + |v| - 1)\Delta(v)$$

or

$$a_v = \Delta(v) - 1 \quad \text{and} \quad c_v = \mathit{first}(v) - 1 \quad \text{and}$$
$$b_v = \mathit{first}(v) - \mathit{last}(v) + n + c(v) - (\mathit{first}(v) + |v| - 1)\Delta(v)$$

In both cases, the tuple (a_v, b_v, c_v, r_v) is easy to obtain. Also, note that $0 \le \Delta(v)-1 \le a_v \le \Delta(v) \le |Occ(v)|$, since $1 \le \Delta(v) \le |Occ(v)|$ by Property 5. □

The following observation is a direct consequence of Lemma 7.

Observation 8. *Given $SST(w)$ with the locus of a factor u of w, one can compute $\mathcal{S}(u)$ in constant time.*

4. Reduction to Two Abstract Problems

Once we have the Seed Suffix Tree $SST(w)$, the LimitedLengthPartialSeeds and PartialSeeds problems can be expressed as relatively simple optimization problems on each edge $path(v)$. Due to the fact that the functions ϕ_v depend on the border array β, we cannot efficiently process each edge independently. Instead, we formulate two abstract problems which involve several queries to be answered off-line.

The following notion encapsulates the property of the border array that we use to solve these problems. We say that an integer array $A[1\,..\,k]$ is a *linear-oscillation* array if

$$\sum_{i=1}^{k-1} |A[i] - A[i+1]| = \mathcal{O}(k).$$

Fact 9. *The border array β is a linear-oscillation array.*

Proof. The conclusion follows from the fact that $0 \le \beta[i+1] \le \beta[i] + 1$ for each $i \in [0\,..\,n-1]$. □

Problems A1 and A2 stated below clearly express the computational task behind the LimitedLengthPartialSeeds and PartialSeeds problems for a string w whose Seed Suffix Tree $SST(w)$ is given.

Problem A1

Input: a linear-oscillation array B of size n and m pairs (ϕ_i, R_i), where ϕ_i is a function $\phi_i(x) = a_i x + b_i + \min(c_i, B[x])$ and $R_i = (\ell_i \,..\, r_i] \subseteq [1 \,..\, n]$ is a non-empty range;

Output: the values $x_i = \operatorname{argmax}\{\phi_i(x) : x \in R_i\}$.[1]

Problem A2

Input: a linear-oscillation array B of size n, a positive integer α, and m pairs (ϕ_i, R_i), where ϕ_i is a function $\phi_i(x) = a_i x + b_i + \min(c_i, B[x])$, and $R_i = (\ell_i \,..\, r_i] \subseteq [1 \,..\, n]$ is a range;

Output: the values $\min\{x \in R_i : \phi_i(x) \geq \alpha\}$.

The following theorems show that indeed the partial seeds problems can be reduced to the two abstract problems. In combination with the results of the following section (Lemmas 17 and 19), they yield the main outcome of our paper: Theorems 2 and 3.

Theorem 10. *Assume that Problem A1 can be solved in $\mathcal{O}(n+m)$ time. Then, given $CST(w)$, the* LimitedLengthPartialSeeds *problem can be solved in $\mathcal{O}(n)$ time.*

Proof. First, we apply Lemma 7 to construct $SST(w)$. We make explicit all nodes corresponding to factors of length $\ell - 1$ and r. This way, each edge either contains only nodes at tree levels in $[\ell \,..\, r]$ or none of them. Note that the functions ϕ_v on the subdivided edges stay the same, only the values r_v change.

Then, we apply the solution for Problem A1 for $B = \beta$ with queries asking to determine for each edge $path(v)$ a node $v_j \in path(v)$ maximizing $\mathcal{S}(v_j)$. Lemma 17 lets us accomplish this in $\mathcal{O}(n)$ time. Taking a global maximum over all edges containing factors of lengths within $[\ell \,..\, r]$, we get the sought factor u which maximizes $\mathcal{S}(u)$ among all factors of w with $|u| \in [\ell \,..\, r]$; see Fig. 4 for an example. □

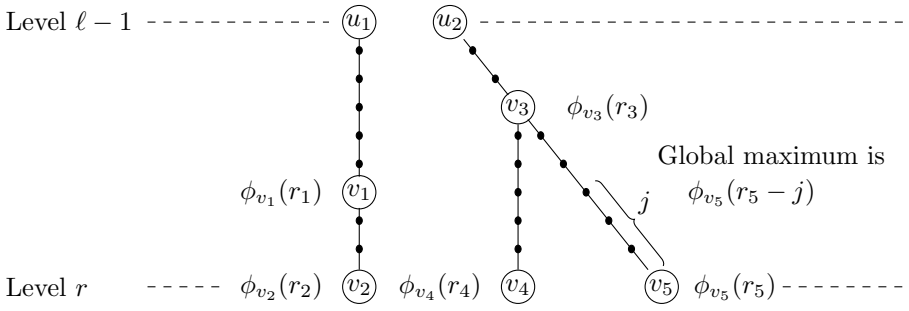

Figure 4: The global maximum over all edges containing factors of lengths within $[\ell \,..\, r]$ is given by $\phi_{v_5}(r_5 - j)$.

Theorem 11. *Assume that Problem A1 can be solved in $\mathcal{O}(n+m)$ time and Problem A2 can be solved in $\mathcal{O}(n + m + \sum_i a_i)$ time. Then, given $CST(w)$, the* PartialSeeds *problem can be solved in $\mathcal{O}(n)$ time.*

Proof. As in the proof of Theorem 10, we start by constructing $SST(w)$ using Lemma 7. The natural next step would be to apply Problem A2 for $B = \beta$ with a query for each edge. However, the algorithm given by

[1] For a set X and a function $f : X \to \mathbb{R}$, we define $\operatorname{argmax}\{f(x) : x \in X\}$ as the largest argument for which f attains its maximum value, that is, $\max\{x \in X : \forall_{x' \in X} f(x) \geq f(x')\}$. We assume $\max \emptyset = -\infty$ and $\min \emptyset = \infty$.

Lemma 19 has an $\mathcal{O}(\sum_i a_i)$ term in the running time, which would become a bottleneck. As a workaround, we reduce the set of queries in an additional preprocessing step.

We say that an edge *path*(v) is *feasible* if $\max\{\mathcal{S}(v_j) : v_j \in \mathit{path}(v)\} \geq \alpha$, and *important* if it is feasible and no ancestor edge is feasible. Observe that all shortest α-partial seeds must lie on important edges. Moreover, note that feasible edges can be detected in linear time using Problem A1 (Lemma 17) applied to compute $\max \mathcal{S}(v_j)$ for each edge *path*(v) of $SST(w)$ (this time we do not introduce any additional extra nodes). Non-important edges are then filtered out with a single scan of the tree.

Next, we solve Problem A2 with $B = \beta$ and a query for each important edge *path*(v). Since $a_v \leq |Occ(v)|$ and no important edge is an ancestor of the other, we have $\sum_v a_v \leq \sum_v |Occ(v)| \leq n$ for these edges, which means that the algorithm of Lemma 19 runs in $\mathcal{O}(n)$ time.

For each edge, we get a single candidate for the shortest α-partial seed. We return the shortest among them as our final answer; see Fig. 5 for an example. □

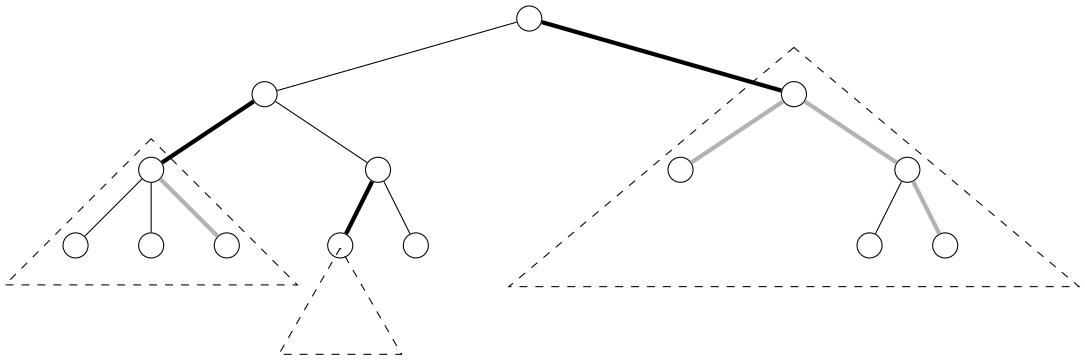

Figure 5: A schematic illustration of the Seed Suffix Tree of a word: black thick edges denote *important* edges; gray thick edges denote *feasible* edges; dashed triangles denote disjoint subtrees rooted at lower endpoints of *important* edges (disjointness of the subtrees implies that $\sum_v |Occ(v)| \leq n$ for those endpoints v).

5. Solutions to Two Abstract Problems

Before we proceed with solutions to Problems A1 and A2, we introduce a few auxiliary definitions and lemmas. For a set $X \subseteq \mathbb{Z}$ and an integer $x \in \mathbb{Z}$, we define $\mathrm{pred}(x, X) = \max\{y \in X : y \leq x\}$, $\mathrm{succ}(x, X) = \min\{y \in X : y \geq x\}$, and $\mathrm{rank}(x, X) = |\{y \in X : y \leq x\}|$.

Lemma 12. *Let $Y[1\,..\,n]$ be an array of integers of magnitude $\mathcal{O}(n)$. For an integer k, let $Y_{\geq k} = \{i : Y[i] \geq k\}$. Given m integer pairs (y_j, k_j), we can compute the values $\mathrm{pred}(y_j, Y_{\geq k_j})$ and $\mathrm{succ}(y_j, Y_{\geq k_j})$ in $\mathcal{O}(n+m)$ time.*

Proof. We reduce the problem to the decremental predecessor problem of maintaining a set $A \subseteq [1\,..\,n]$, initially equal to $[1\,..\,n]$, subject to two operations:

- $A := A \setminus \{x\}$,
- compute $\mathrm{succ}(x, A)$ and $\mathrm{pred}(x, A)$.

Note that the decremental predecessor problem can be solved using a variant of the Union-Find data structure, where the universe $[1\,..\,n]$ is always partitioned into intervals. (Here, indices i and i' are in the same class if and only if $\mathrm{pred}(i, A) = \mathrm{pred}(i', A)$.) Gabow and Tarjan [17] presented an on-line solution of this variant, which processes any sequence of m operations in $\mathcal{O}(n + m)$ time in total.

We maintain $A = Y_{\geq k}$ for increasing values of k. For each value, we first answer queries with $k_j = k$ and then remove from A all indices i such that $Y[i] = k$. We sort both queries and entries of Y before, so

that we handle only values k, for which there is a query with $k = k_j$ or an entry with $Y[i] = k$. Since these values are integers of magnitude $\mathcal{O}(n)$, sorting takes $\mathcal{O}(n + m)$ time, and the main part of the algorithm, using the data structure for decremental predecessor queries, also takes $\mathcal{O}(n + m)$ time. □

Lemma 13. *Assume we are given a family $\{X_a \subseteq [1\,..\,n] : a \in [1\,..\,n]\}$ of sets of total size $\mathcal{O}(n)$ and functions $\psi_a : [1\,..\,n] \to \mathbb{Z}$, each computable in constant time. Given m pairs (a_j, R_j) or m triplets (a_j, β_j, R_j), where $a_j \in [1\,..\,n]$, $R_j \subseteq [1\,..\,n]$ is an interval, and $\beta_j \in \mathbb{Z}$, we can compute*

$$\operatorname{argmax}\{\psi_{a_j}(x) \,:\, x \in X_{a_j} \cap R_j\} \text{ in case of pairs, or}$$

$$\min\{x \in X_{a_j} \cap R_j \,:\, \psi_{a_j}(x) \geq \beta_j\} \text{ in case of triples}$$

offline in $\mathcal{O}(n + m)$ time.

Proof. For each $a \in [1\,..\,n]$, we construct an array T_a of size $|X_a|$ which stores values $\psi_a(x)$ for all $x \in X_a$ in the increasing order of x. Then for $x \in X_a$ we have $\psi_a(x) = T_a[\operatorname{rank}(x, X_a)]$. Each range $R_j = (\ell_j\,..\,r_j]$ is translated to a range

$$R'_j = (\operatorname{rank}(\ell_j, X_{a_j})\,..\,\operatorname{rank}(r_j, X_{a_j})].$$

This can be done in $\mathcal{O}(n + m)$ time due to the following claim:

Claim 14. *We can compute off-line in $\mathcal{O}(n + k)$ time k values*

$$\operatorname{rank}(y_j, X_{b_j}), \quad \text{for } y_j, b_j \in [1\,..\,n].$$

Proof (of the claim). We store an array *counter*$[a]$, $a \in [1\,..\,n]$. Initially the array contains only zeros. For each $x \in [1\,..\,n]$, we increment *counter*$[a]$ for all $a \in [1\,..\,n]$ such that $x \in X_a$. Thus, just after processing x, *counter*$[a]$ becomes $\operatorname{rank}(x, X_a)$. This way we can answer queries provided that we previously sort them by y_j. □

Now, the original queries reduce to queries concerning R'_j for T_a. Queries of the first type become range minimum queries (and thus can be answered on-line in constant time after linear-time preprocessing; see [18, 19]), while queries of the second type become the successor (succ) queries considered in Lemma 12 for $Y = T_a$ and $k_j = \beta_j$. Since the values in T_a are not necessarily of magnitude $\mathcal{O}(|X_a|)$, we need to simultaneously process all arrays T_a, jointly sorting entries and queries. Otherwise, the algorithm of Lemma 12 remains unchanged. □

The following simple result is the reason behind the linear-oscillation assumption in both problems.

Observation 15. *For a linear-oscillation array B of size n and a non-negative integer a, let $F_a = \{x : B[x + 1] < B[x] - a\}$. Then $\sum_{a=0}^{\infty} |F_a| = \mathcal{O}(n)$.*

Proof. Each $x \in [1\,..\,n - 1]$ belongs to F_a if and only if $a < B[x] - B[x + 1]$. Hence, each x belongs to at most $B[x] - B[x + 1]$ sets F_a. The sum is, therefore, bounded by the total decrease of B, which is $\mathcal{O}(n)$ for a linear-oscillation array. □

5.1. Solution for Problem A1

First, we solve the following simplified version of Problem A1:

> PROBLEM B1
>
> **Input:** a linear-oscillation array B of size n and m pairs (ϕ_i, R_i), where ϕ_i is a function $\phi_i(x) = a_i x + B[x]$ and $R_i = (\ell_i\,..\,r_i] \subseteq [1\,..\,n]$ is a non-empty range;
>
> **Output:** the values $x_i = \operatorname{argmax}\{\phi_i(x) : x \in R_i\}$.

Lemma 16. *Problem B1 can be solved in $\mathcal{O}(n + m)$ time.*

Proof. Recall the sets F_a defined in Observation 15. Note that $x_i \in F_{a_i}$ or $x_i = r_i$. Indeed, if $x \notin F_{a_i}$ and $x \neq r_i$, then $x+1 \in R_i$ and

$$\phi_i(x+1) = a_i(x+1) + B[x+1] \geq a_i x + a_i + B[x] - a_i = \phi_i(x).$$

Consequently, it is enough to compute

$$x'_i = \operatorname{argmax}\{\phi_i(x) : x \in F_{a_i} \cap R_i\}.$$

By Lemma 13, the values x'_i can be computed in $\mathcal{O}(n+m)$ time. We then have $x_i \in \{x'_i, r_i\}$, and it is easy to check in constant time which of the two candidates x_i actually is. □

Our solution to Problem A1 uses Lemma 16 as a black box:

Lemma 17. *Problem A1 can be solved in $\mathcal{O}(n+m)$ time.*

Proof. Let

$$y_i = \max(\{x \in R_i : B[x] \geq c_i\} \cup \{\ell_i\}).$$

Observe that every $x \in R_i$ such that $x \leq y_i$ satisfies $\phi_i(x) \leq \phi_i(y_i)$. Indeed, such x exists only if $\ell_i < y_i$, which implies $B[y_i] \geq c_i$, and thus

$$\phi_i(x) = a_i x + b_i + \min(c_i, B[x]) \leq a_i y_i + b_i + c_i = \phi_i(y_i).$$

Consequently,

$$x_i = \operatorname{argmax}\{\phi_i(x) : x \in R_i\} \geq y_i.$$

Note that for $x \in R_i$ such that $x > y_i$ we have $\phi_i(x) = a_i x + b_i + B[x]$. Thus, it suffices to solve Problem B1 for $R'_i = (y_i \,..\, r_i]$ (if $R'_i \neq \emptyset$). We apply Lemma 16, which yields an $\mathcal{O}(n+m)$-time algorithm. This way we find:

$$x'_i = \operatorname{argmax}\{a_i x + B[x] : x \in R'_i\}.$$

We must have $x_i \in \{x'_i, y_i\}$, and it is easy to check in constant time which of the two candidates x_i is.

The missing step of the algorithm is how to determine the values y_i. We claim that they can be computed off-line in $\mathcal{O}(n+m)$ time. First, we shall compute

$$y'_i = \max\{x \leq r_i : B[x] \geq c_i\} = \operatorname{pred}(r_i, B_{\geq c_i})$$

using Lemma 12. Then y_i can be expressed as $\max(y'_i, \ell_i)$. □

5.2. Solution for Problem A2

Like before, we start with a simplified version of the problem:

> PROBLEM B2
>
> **Input:** a linear-oscillation array B of size n and m triples (ϕ_i, α_i, R_i), where $\phi_i(x) = a_i x + B[x]$, α_i is a positive integer and $R_i = (\ell_i \,..\, r_i] \subseteq [1 \,..\, n]$ is a range;
>
> **Output:** the values $x_i = \min\{x \in R_i : \phi_i(x) \geq \alpha_i\}$.

Lemma 18. *Problem B2 can be solved in $\mathcal{O}(n + m + \sum_i a_i)$ time.*

Proof. Before we proceed with the algorithm, let us prove a few properties of the sought values x_i (provided that $x_i \neq \infty$). Let us recall the sets F_a from Observation 15 and define ranges $R'_i = (\ell'_i \,..\, r'_i]$ where

$$r'_i = \min(r_i, \operatorname{succ}(x_i, F_{a_i})), \quad \text{and} \quad \ell'_i = \max(\ell_i, \operatorname{pred}(x_i - 1, F_{a_i})).$$

As noted in the proof of Lemma 16, we have $\phi_i(x+1) \geq \phi_i(x)$ if $x \notin F_{a_i}$, so ϕ_i is non-decreasing within $R'_i \subseteq R_i$. In particular, $\phi_i(r'_i) \geq \alpha_i$. Since x_i is the leftmost index $x \in R_i$ with $\phi_i(x) \geq \alpha_i$, we actually have $r'_i = x'_i$ for

$$x'_i = \min(\{x \in R_i \cap F_{a_i} : \phi_i(x) \geq \alpha_i\} \cup \{r_i\}).$$

These values can be computed off-line in $\mathcal{O}(n+m)$ time by Lemma 13. Observe that $x_i = \infty$ if and only if $\phi'_i(x'_i) < \alpha_i$, which is easy to detect. Hence, we may already assume that $x_i \neq \infty$, which means that r'_i is well defined and equal to x'_i. Now, the value ℓ'_i can be easily computed due to the fact that

$$\ell'_i = \max(\ell_i, \mathrm{pred}(r'_i - 1, F_{a_i}))$$

Once we have the interval R'_i, we can exploit monotonicity of ϕ_i to determine the value x_i in $\mathcal{O}(\log |R'_i|)$ time using binary search.

However, this might be too slow if $|R'_i|$ is large. Therefore, in the construction above we shall replace sets F_a with sets G_a defined as

$$G_a = F_a \cup \{x \in [1\,..\,n] \,:\, x \bmod 2^a = 0\}$$

so that we can guarantee that $|R'_i| \leq 2^{a_i}$. Observation 15 and the fact that $\sum_{a=0}^{\infty} \frac{1}{2^a} = \mathcal{O}(1)$ imply

$$\sum_{a=0}^{\infty} |G_a| \leq \sum_{a=0}^{\infty} |F_a| + \sum_{a=0}^{\infty} \tfrac{n}{2^a} = \mathcal{O}(n).$$

Also, note that simple arithmetics can be used to compute $\mathrm{pred}(x, G_a)$ and $\mathrm{succ}(x, G_a)$ based on $\mathrm{pred}(x, F_a)$ and $\mathrm{succ}(x, F_a)$. Thus, the sets G_a can indeed be used instead of F_a without any slowdown. The total running time then becomes $\mathcal{O}(n + m + \sum_i \log |R'_i|) = \mathcal{O}(n + m + \sum_i a_i)$. □

Next, we solve Problem A2 using a simple reduction to Problem B2.

Lemma 19. *Problem A2 can be solved in $\mathcal{O}(n + m + \sum_i a_i)$ time.*

Proof. We set $\alpha_i = \alpha - b_i$ and let $y_i = \left\lceil \frac{\alpha_i - c_i}{a_i} \right\rceil$. For $a_i = 0$, we set $y_i = \infty$ if $c_i < \alpha_i$ and $y_i = -\infty$ otherwise. Note that for $x \in R_i$ such that $x < y_i$, we have $a_i x + c_i < \alpha_i$, so $\phi_i(x) < \alpha$. Therefore $x_i \geq y_i$. On the other hand, if $x \geq y_i$, then $a_i x + c_i \geq \alpha_i$, so $\phi_i(x) \geq \alpha$ if and only if $a_i x + B[x] \geq \alpha_i$. Consequently, it suffices to solve Problem B2 with $R'_i = R_i \cap [y_i\,..\,\infty)$. □

6. Conclusions

We introduced a notion of approximate quasiperiodicity called partial seed. To compute partial seeds in a word, we applied an augmented version of the suffix tree based on a similar data structure that was designed for previously studied partial covers. It allows computation of partial seeds in $\mathcal{O}(n \log n)$ time (more precisely, in $\mathcal{O}(n)$ time plus the construction time of the data structure for partial covers).

An interesting open question is whether one can compute the shortest α-partial seed for each $\alpha \in [1\,..\,n]$ faster than applying Theorem 3 for each α (which works in $\mathcal{O}(n^2)$ time). An analogous problem for partial covers is known to have a non-trivial $\mathcal{O}(n \log n)$-time solution [13].

Acknowledgements

Tomasz Kociumaka is supported by Polish budget funds for science in 2013–2017 as a research project under the 'Diamond Grant' program (Ministry of Science and Higher Education, Republic of Poland, grant number DI2012 01794). Jakub Radoszewski receives financial support of Foundation for Polish Science and is supported by the Polish Ministry of Science and Higher Education under the 'Iuventus Plus' program in 2015-2016 grant no. 0392/IP3/2015/73. Wojciech Rytter is supported by grant no. NCN2014/13/B/ST6/00770 of the National Science Centre.

References

[1] T. Kociumaka, S. P. Pissis, J. Radoszewski, W. Rytter, T. Waleń, Efficient algorithms for shortest partial seeds in words, in: A. S. Kulikov, S. O. Kuznetsov, P. A. Pevzner (Eds.), Combinatorial Pattern Matching, CPM 2014, Vol. 8486 of LNCS, Springer, 2014, pp. 192–201. doi:10.1007/978-3-319-07566-2_20.

[2] M. Crochemore, L. Ilie, W. Rytter, Repetitions in strings: Algorithms and combinatorics, Theor. Comput. Sci. 410 (50) (2009) 5227–5235. doi:10.1016/j.tcs.2009.08.024.

[3] A. Apostolico, A. Ehrenfeucht, Efficient detection of quasiperiodicities in strings, Theor. Comput. Sci. 119 (2) (1993) 247–265. doi:10.1016/0304-3975(93)90159-Q.

[4] A. Apostolico, M. Farach, C. S. Iliopoulos, Optimal superprimitivity testing for strings, Inf. Process. Lett. 39 (1) (1991) 17–20. doi:10.1016/0020-0190(91)90056-N.

[5] D. Moore, W. F. Smyth, An optimal algorithm to compute all the covers of a string, Inf. Process. Lett. 50 (5) (1994) 239–246. doi:10.1016/0020-0190(94)00045-X.

[6] D. Moore, W. F. Smyth, A correction to "An optimal algorithm to compute all the covers of a string", Inf. Process. Lett. 54 (2) (1995) 101–103. doi:10.1016/0020-0190(94)00235-Q.

[7] D. Breslauer, An on-line string superprimitivity test, Inf. Process. Lett. 44 (6) (1992) 345–347. doi:10.1016/0020-0190(92)90111-8.

[8] Y. Li, W. F. Smyth, Computing the cover array in linear time, Algorithmica 32 (1) (2002) 95–106. doi:10.1007/s00453-001-0062-2.

[9] C. S. Iliopoulos, D. W. G. Moore, K. Park, Covering a string, Algorithmica 16 (3) (1996) 288–297. doi:10.1007/BF01955677.

[10] T. Kociumaka, M. Kubica, J. Radoszewski, W. Rytter, T. Waleń, A linear time algorithm for seeds computation, in: Y. Rabani (Ed.), 23rd Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, SIAM, 2012, pp. 1095–1112. doi:10.1137/1.9781611973099.

[11] J. S. Sim, K. Park, S. Kim, J. Lee, Finding approximate covers of strings, Journal of Korea Information Science Society 29 (1) (2002) 16–21.

[12] M. Christodoulakis, C. S. Iliopoulos, K. Park, J. S. Sim, Approximate seeds of strings, Journal of Automata, Languages and Combinatorics 10 (5/6) (2005) 609–626.

[13] T. Kociumaka, S. P. Pissis, J. Radoszewski, W. Rytter, T. Waleń, Fast algorithm for partial covers in words, Algorithmica 73 (1) (2015) 217–233. doi:10.1007/s00453-014-9915-3.

[14] T. Flouri, C. S. Iliopoulos, T. Kociumaka, S. P. Pissis, S. J. Puglisi, W. F. Smyth, W. Tyczyński, Enhanced string covering, Theor. Comput. Sci. 506 (2013) 102–114. doi:10.1016/j.tcs.2013.08.013.

[15] M. Crochemore, W. Rytter, Jewels of Stringology, World Scientific, 2003.

[16] M. Crochemore, C. Hancart, T. Lecroq, Algorithms on Strings, Cambridge University Press, New York, NY, USA, 2007.

[17] H. N. Gabow, R. E. Tarjan, A linear-time algorithm for a special case of disjoint set union, J. Comput. Syst. Sci. 30 (2) (1985) 209–221.

[18] D. Harel, R. E. Tarjan, Fast algorithms for finding nearest common ancestors, SIAM J. Comput. 13 (2) (1984) 338–355. doi:10.1137/0213024.

[19] M. A. Bender, M. Farach-Colton, The LCA problem revisited, in: G. H. Gonnet, D. Panario, A. Viola (Eds.), Latin American Symposium on Theoretical Informatics, LATIN 2000, Vol. 1776 of LNCS, Springer Berlin Heidelberg, 2000, pp. 88–94. doi:10.1007/10719839_9.